Neural Ordinary Differential Equations

Krishna Kumar

University of Texas at Austin

krishnak@utexas.edu

Overview

Learning Objectives

- Understand the connection between ResNets and continuous dynamics
- Master the Neural ODE framework and adjoint method
- Implement ODENets for image classification
- Apply continuous normalizing flows for generative modeling
- Build latent ODE models for irregular time series

▸ Open Notebook

The ResNet Formula

A residual network transforms hidden states layer by layer:

$$h_{t+1} = h_t + f(h_t, \theta_t) \tag{1}$$

where $t \in \{0, 1, \ldots, T\}$ indexes the layers.

The Key Question

What happens as we add more layers ($T \to \infty$) and take smaller steps?

Let's understand why this question matters. In a ResNet, we're adding a small correction $f(h_t, \theta_t)$ to our hidden state. Think of this like taking steps on a path: we're at position h_t, and we move by some amount $f(h_t, \theta_t)$.

Now imagine we make our layers shallower – each one does less work. To compensate, we add more layers. In the limit, each layer does an infinitesimally small update, but we have infinitely many of them.

This is exactly the setup for calculus! When we have many small discrete steps, we get a continuous process. The transformation becomes: $h_{t+1} - h_t = f(h_t, \theta_t)$. If we divide by the step size $\Delta t = 1$ and take the limit as steps become infinitesimal, we get $\frac{dh}{dt} = f(h, \theta)$.

This connection means: ResNets with many layers $\approx$ following a continuous path through hidden state space.

The Euler Connection

The ResNet update is the **Euler discretization** of an ODE:

$$\frac{dh(t)}{dt} = f(h(t), t, \theta) \tag{2}$$

Key Insight

A ResNet with infinitely many infinitesimal layers $\equiv$ solving an ODE

Instead of specifying discrete layers, we parameterize the **derivative** of the hidden state using a neural network.

Here's the beautiful shift in perspective. In traditional neural networks, we explicitly define what happens at each layer. In Neural ODEs, we instead define *how the hidden state changes*.

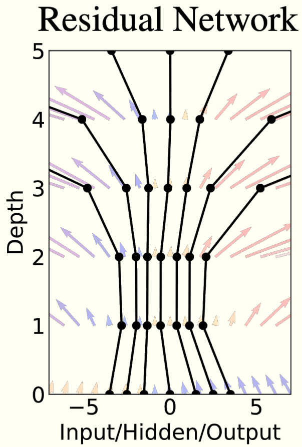
Think of it like describing motion: instead of saying "at time 1 you're at position x_1, at time 2 you're at position x_2," we say "your velocity is $v(x, t)$" and let calculus figure out where you'll be.

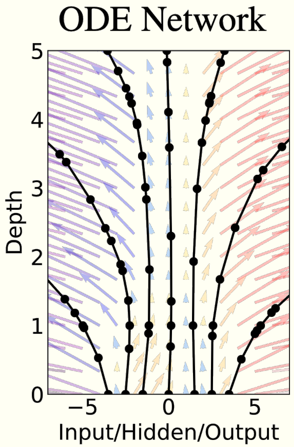
The function $f(h(t), t, \theta)$ is our neural network. It takes the current hidden state and time, and outputs the instantaneous rate of change. To get the hidden state at any time t_1, we solve:

$$h(t_1) = h(t_0) + \int_{t_0}^{t_1} f(h(t), t, \theta)\, dt$$

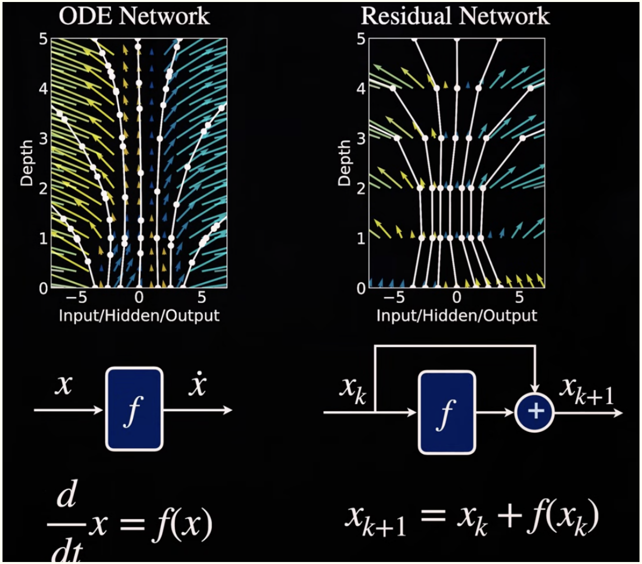
This is fundamentally different from ResNets: the network depth becomes a continuous quantity. We're not limited to discrete layers anymore. The ODE solver adaptively determines how many "function evaluations" are needed for a given accuracy tolerance.

ResNet vs ODE





ResNet vs ODE



$$\frac{d}{dt}x = f(x)$$

$$x_{k+1} = x_k + f(x_k)$$

Numerical Integration: From Discrete to Continuous

The Mathematician's View:
The exact solution to $\frac{dx}{dt} = f(x, t, \theta)$ from time t_k to t_{k+1} is:

$$x_{k+1} = x_k + \int_{t=t_k}^{t=t_{k+1}} f(x(\tau), \tau, \theta)\, d\tau \tag{3}$$

The Problem: We can't compute this integral analytically!

The Solution: Numerical integration schemes approximate this integral

- **Euler (ResNet):** Simplest approximation, worst accuracy
- **Runge-Kutta:** Better approximations, higher accuracy
- **Adaptive methods:** Adjust step size automatically

Understanding the Integral: When we write $x_{k+1} = x_k + \int_{t_k}^{t_{k+1}} f(x(\tau), \tau, \theta) d\tau$, we're saying: "To get from x_k to x_{k+1}, we need to accumulate all the infinitesimal changes $f(x(\tau), \tau) d\tau$ along the entire trajectory from time t_k to t_{k+1}."
The variable τ is a dummy integration variable that runs from t_k to t_{k+1}. At each instant τ, we evaluate the vector field f at the current state $x(\tau)$.
Why Numerical Integration? For most neural networks f, this integral has no closed-form solution. We must approximate it numerically.
The Key Tradeoff: More accurate integrators require more function evaluations of f, which means more forward passes through the neural network. But they give better approximations of the true solution.
Neural ODE's Advantage: Because we're approximating the continuous integral, we can choose different integration schemes based on our needs:

- Fast but rough? Use Euler
- Accurate? Use RK4 or Dormand-Prince
- Need energy conservation? Use symplectic integrators

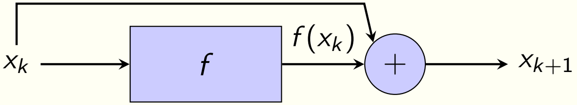
ResNets are stuck with Euler (one step, $\Delta t = 1$), but Neural ODEs can use any integrator!

Integration Schemes: Euler (ResNet)

Euler Method (Forward Euler):

$$x_{k+1} = x_k + \Delta t \cdot f(x_k, t_k, \theta) \tag{4}$$

With $\Delta t = 1$: $x_{k+1} = x_k + f(x_k, \theta) \quad \leftarrow$ ResNet!



Properties:

- One function evaluation per step
- First-order accurate: error $\mathcal{O}(\Delta t^2)$
- Simple but can be unstable

Euler Method Explained: The simplest approximation of $\int_{t_k}^{t_{k+1}} f(x(\tau))d\tau$ is to assume f is constant over the interval:

$$\int_{t_k}^{t_{k+1}} f(x(\tau))d\tau \approx f(x_k) \cdot (t_{k+1} - t_k) = f(x_k) \cdot \Delta t$$

This is like approximating the area under a curve with a single rectangle (left Riemann sum).

The Block Diagram:

- Input x_k goes into function f (the neural network)
- Output $f(x_k)$ is the instantaneous rate of change
- Add this change to the current state x_k
- Result is $x_{k+1} = x_k + f(x_k)$

Why ResNets Use This: ResNets weren't designed for differential equations. The residual connection $x_{k+1} = x_k + f(x_k)$ just happens to be Euler's method with $\Delta t = 1$.

The Problem with Euler:

- Large error accumulation over many steps
- Can become unstable for stiff equations
- First-order accuracy means doubling steps only halves the error

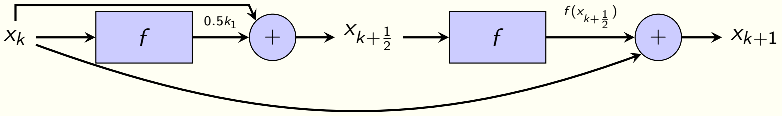
For Neural ODEs, we can do much better!

Integration Schemes: Midpoint (2nd Order)

Midpoint Method (2nd order Runge-Kutta):

$$k_1 = f(x_k, t_k, \theta) \tag{5}$$

$$x_{k+1} = x_k + \Delta t \cdot f(x_k + \frac{\Delta t}{2} k_1, t_k + \frac{\Delta t}{2}, \theta) \tag{6}$$



Properties:

- Two function evaluations per step
- Second-order accurate: error $\mathcal{O}(\Delta t^3)$
- Much more accurate than Euler for same Δt

Midpoint Method Strategy: Instead of using $f(x_k)$ over the entire interval (like Euler), we:

1. Take a half-step using Euler: $x_{k+1/2} = x_k + \frac{\Delta t}{2} f(x_k)$
2. Evaluate f at this midpoint: $f(x_{k+1/2})$
3. Use this midpoint slope for the full step: $x_{k+1} = x_k + \Delta t \cdot f(x_{k+1/2})$

This is like approximating the integral with a rectangle centered at the midpoint (midpoint Riemann sum), which is much more accurate!

The Block Diagram Walkthrough:

- First block: Evaluate $f(x_k)$ to get k_1
- First sum: Compute $x_k + 0.5\Delta t \cdot k_1$ (half-step Euler)
- Second block: Evaluate f at the midpoint
- Second sum: Add this to x_k to get x_{k+1}

Why This is Better:

- Evaluates f at a point that better represents the average over $[t_k, t_{k+1}]$
- Second-order accuracy: halving Δt reduces error by factor of 4 (not 2!)
- Only costs 2 function evaluations vs. 1 for Euler

Integration Schemes: RK4 (4th Order)

Classical 4th Order Runge-Kutta (RK4):

$$k_1 = f(x_k, t_k, \theta) \tag{7}$$

$$k_2 = f(x_k + \frac{\Delta t}{2} k_1, t_k + \frac{\Delta t}{2}, \theta) \tag{8}$$

$$k_3 = f(x_k + \frac{\Delta t}{2} k_2, t_k + \frac{\Delta t}{2}, \theta) \tag{9}$$

$$k_4 = f(x_k + \Delta t \cdot k_3, t_k + \Delta t, \theta) \tag{10}$$

$$x_{k+1} = x_k + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{11}$$

Properties:

- Four function evaluations per step
- Fourth-order accurate: error $\mathcal{O}(\Delta t^5)$
- Industry standard for non-stiff ODEs
- Much more stable than Euler/Midpoint

RK4 Strategy: The most famous Runge-Kutta method! It uses a weighted average of four slope estimates:

- k_1: slope at the beginning of the interval
- k_2: slope at the midpoint, using k_1 to get there
- k_3: slope at the midpoint, using k_2 to get there (better estimate!)
- k_4: slope at the end, using k_3 to get there

Then combines them with weights $\frac{1}{6}[k_1 + 2k_2 + 2k_3 + k_4]$ (Simpson's rule!).
Intuition: This is like approximating the integral with Simpson's rule - using a weighted average of slopes at carefully chosen points to get a very accurate approximation of the area under the curve.
Why Fourth-Order Matters:

- Halving Δt reduces error by factor of 16 (not 2 or 4!)
- Can use much larger time steps than Euler for same accuracy
- Very stable for most problems

Cost vs. Benefit:

- Costs $4\times$ Euler per step
- But often can take steps $10\times$ larger for same accuracy
- Net result: much fewer total evaluations!

ResNet vs Neural ODE

Residual Network

- Discrete transformations
- Fixed number of layers T
- $x_{k+1} = x_k + f(x_k, \theta_k)$
- Memory: $\mathcal{O}(T)$
- Evenly spaced time steps

Neural ODE

- Continuous dynamics
- Adaptive depth
- $\frac{dx}{dt} = f(x(t), t, \theta)$
- Memory: $\mathcal{O}(1)$
- Irregular time steps OK

Key Insights: ResNets are essentially doing Euler integration with a time step of $\Delta t = 1$. And Euler integration is about the WORST way to integrate a differential equation! It's quick and dirty, but highly prone to instability and huge errors.

Why This Matters: If ResNets have been so successful based on this simple (and crude) idea, imagine what we can do with better numerical integrators!

The ResNet Problem:

- Models one large time step: $x_{k+1} = x_k + f(x_k)$
- Requires evenly spaced data
- Uses worst possible integrator (Euler)
- Each layer is a fixed discretization

The Neural ODE Solution:

- Models the actual vector field: $\frac{dx}{dt} = f(x, t)$
- Works with irregularly spaced data
- Uses sophisticated integrators (Dormand-Prince, RK4, etc.)
- Adaptive computation based on solver tolerance

Analogy: It's like the difference between compound interest yearly vs. continuously. Continuous compounding is just better!

Basic Neural ODE Components

1. ODE Function $f(h, t, \theta)$

Neural network that computes the derivative $\frac{dh}{dt}$

2. ODE Solver

Integrates $h(t)$ from t_0 to t_1:

$$h(t_1) = h(t_0) + \int_{t_0}^{t_1} f(h(t), t, \theta) dt \tag{12}$$

3. Adjoint Method

Computes gradients $\frac{\partial L}{\partial \theta}$ efficiently

Component 1 – The ODE Function: This is just a neural network! Typically a small MLP with 2-3 hidden layers. Input: current hidden state $h(t)$ and possibly time t. Output: the rate of change $\frac{dh}{dt}$.
Example architecture: $h \in \mathbb{R}^{64}$, then f might be Linear(64, 128) $\rightarrow$ Tanh $\rightarrow$ Linear(128, 64). The output dimension must match the hidden state dimension.

Component 2 – The ODE Solver: We use adaptive solvers like Dormand-Prince (dopri5) or Adams methods. These automatically adjust step sizes to maintain accuracy. Tolerances `rtol` and `atol` control the trade-off between speed and precision. Typical values: `rtol=1e-3`, `atol=1e-4`.
The solver calls $f(h(t), t, \theta)$ many times to numerically integrate the ODE. This is why memory efficiency matters – we don't want to store gradients for every evaluation!

Component 3 – The Adjoint Method: This is the secret sauce. Instead of backpropagating through every ODE solver step, we solve a second ODE backward in time to compute exact gradients. The `torchdiffeq` library implements this with `odeint_adjoint`.

Memory Efficiency: The Adjoint Method

Standard Backpropagation:

- Store all intermediate layer activations
- Memory: $\mathcal{O}(L)$ where L = number of layers

Adjoint Method:

- Solve a second ODE backwards in time
- Recompute forward states during backward pass
- Memory: $\mathcal{O}(1)$ independent of depth

The adjoint state $a(t) = \frac{\partial L}{\partial h(t)}$ evolves as:

$$\frac{da(t)}{dt} = -a(t)^T \frac{\partial f(h(t), t, \theta)}{\partial h} \tag{13}$$

The Problem: To train the network, we need gradients $\frac{\partial L}{\partial \theta}$. Standard backprop would store every intermediate state $h(t)$ that the ODE solver computed. For adaptive solvers, this could be hundreds of evaluations!
The Solution (Adjoint Sensitivity): Define the adjoint state $a(t) = \frac{\partial L}{\partial h(t)}$. This tracks how the loss changes with respect to the hidden state at time t.
Key Derivation: Start with the chain rule. The loss L depends on the final state $h(t_1)$, which depends on all intermediate states. By the chain rule:

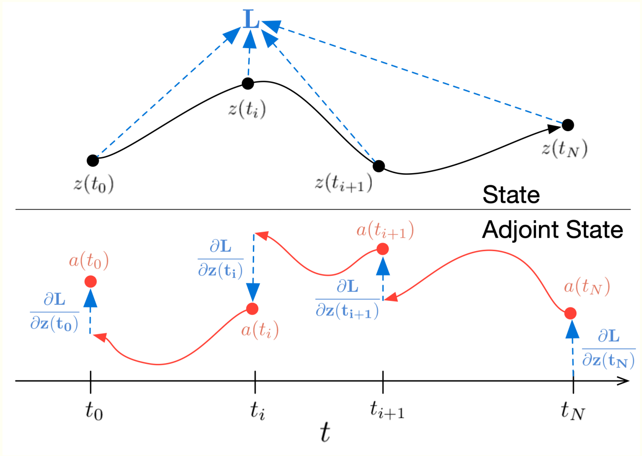
$$\frac{dL}{d\theta} = \int_{t_0}^{t_1} \frac{\partial L}{\partial h(t)} \frac{\partial h(t)}{\partial \theta} dt$$

The Magic: Instead of computing $\frac{\partial h(t)}{\partial \theta}$ forward in time (which requires storing everything), we compute $a(t) = \frac{\partial L}{\partial h(t)}$ backward in time by solving another ODE! **How $a(t)$ evolves:** Consider how $a(t)$ changes. At time $t + \Delta t$, we have $h(t + \Delta t) = h(t) + f(h(t), t, \theta)\Delta t$. By chain rule:

$$a(t) = a(t + \Delta t) + a(t + \Delta t)^T \frac{\partial f}{\partial h} \Delta t$$

Taking the limit as $\Delta t \to 0$: $\frac{da(t)}{dt} = -a(t)^T \frac{\partial f}{\partial h}$. The negative sign comes from integrating backward in time.
Getting gradients for parameters: Similarly, the gradient with respect to parameters is:

The Adjoint Method Visualized


L
$z(t_0)$
$z(t_i)$
$z(t_{i+1})$
$z(t_N)$
State
Adjoint State
$a(t_0)$
$\frac{\partial L}{\partial z(t_0)}$
$\frac{\partial L}{\partial z(t_i)}$
$a(t_i)$
$a(t_{i+1})$
$\frac{\partial L}{\partial z(t_{i+1})}$
$a(t_N)$
$\frac{\partial L}{\partial z(t_N)}$
t_0
t_i
t_{i+1}
t_N
t

- **Forward**: Solve ODE from t_0 to t_1
- **Backward**: Solve augmented ODE from t_1 to t_0
- Automatically handled by `odeint_adjoint`

This diagram shows the complete forward-backward process. The forward pass (blue) solves the ODE to get $h(t_1)$ and computes the loss. The backward pass (red) solves a *different* ODE going backward in time to compute all gradients.

Notice: we don't store any intermediate states from the forward pass! During the backward pass, we recompute $h(t)$ on-the-fly as needed. This is the key to $\mathcal{O}(1)$ memory.

The `torchdiffeq` library provides `odeint_adjoint` which automatically implements this algorithm. Use it instead of `odeint` for training to get memory savings.

The Training Challenge

What we're doing: Learning the vector field $f(x, t, \theta)$

The Process:

1. Tweak parameters θ of the neural network
2. Numerically integrate along the vector field
3. Compare integrated trajectory to observed data
4. Minimize loss between prediction and data

Key Challenge: Hidden states between observations
Data is sampled at discrete times: $t_0, t_1, t_2, \dots$
But the trajectory flows continuously: $x(\tau)$ for all $\tau \in [t_0, t_1]$

The Question

How do we compute gradients through the ODE solver without storing all intermediate states?

When training a Neural ODE, we need to compute $\frac{\partial L}{\partial \theta}$ where L is our loss function and θ are the network parameters.

The Hidden State Problem: Between any two measurement points, there's a continuous trajectory. We call this the "hidden state" $x(\tau)$ where τ is continuous time between measurements.

When we integrate from t_0 to $t_0 + \Delta t$, we traverse this hidden state:

$$x(t_0 + \Delta t) = \Phi(x(t_0), t_0, \Delta t, \theta)$$

where Φ is the flow map that integrates the ODE.

Why This Matters: Our loss function L depends on $x(t)$ at observation times, but $x(t)$ depends on this entire continuous hidden trajectory. To train θ, we need:

$$\frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial x(t)} \frac{\partial x(t)}{\partial \theta}$$

But computing $\frac{\partial x(t)}{\partial \theta}$ requires tracking how parameters affect the entire continuous trajectory!

Two Bad Options:

1. Store every intermediate state $\rightarrow \mathcal{O}(N)$ memory (unacceptable)
2. Compute finite differences $\rightarrow$ very inaccurate and expensive

The Good Option: Use the adjoint method with automatic differentia-

Algorithm 1: Adjoint Sensitivity Method

Input: Dynamics f, loss L, initial state $h(t_0)$, times $t_0 < t_1$

Forward Pass:

1. Solve ODE: $h(t_1) = h(t_0) + \int_{t_0}^{t_1} f(h(t), t, \theta)\, dt$
2. Compute loss: $L = L(h(t_1))$

Backward Pass: Define augmented state $s(t) = [a(t), \frac{\partial L}{\partial \theta}(t), \frac{\partial L}{\partial t_0}(t)]$

Initialize: $a(t_1) = \frac{\partial L}{\partial h(t_1)}$, others zero

Solve augmented ODE backward from t_1 to t_0:

$$\frac{da(t)}{dt} = -a(t)^T \frac{\partial f(h(t), t, \theta)}{\partial h}$$
$$\frac{d}{dt}\frac{\partial L}{\partial \theta} = -a(t)^T \frac{\partial f(h(t), t, \theta)}{\partial \theta}$$
$$\frac{d}{dt}\frac{\partial L}{\partial t_0} = -a(t)^T f(h(t), t, \theta)$$

Why this works: The adjoint method comes from optimal control theory. We're computing gradients of an integral functional.
Forward Pass (Step 1-2): Standard. Run the ODE solver to get the final state and compute the loss. Nothing special here.
Backward Pass Setup: We want three gradients: (1) $\frac{\partial L}{\partial \theta}$ to update parameters, (2) $\frac{\partial L}{\partial t_0}$ if initial time is learned, (3) adjoint $a(t) = \frac{\partial L}{\partial h(t)}$ to propagate gradients through time.
The Augmented State: We pack all three quantities into a single vector $s(t)$ and solve one big ODE for all of them simultaneously. This is computationally efficient.
Initialization: At $t = t_1$ (end of forward pass), we know $\frac{\partial L}{\partial h(t_1)}$ from automatic differentiation. The gradient w.r.t. parameters and initial time start at zero and accumulate as we integrate backward. **The Dynamics:** Each component has its own ODE. Let's understand each:
Adjoint evolution: $\frac{da}{dt} = -a^T \frac{\partial f}{\partial h}$. This propagates the gradient backward through the dynamics. The Jacobian $\frac{\partial f}{\partial h}$ tells us how changes in h affect the derivative f. The negative sign comes from integrating backward in time.
Parameter gradients: $\frac{d}{dt}\frac{\partial L}{\partial \theta} = -a^T \frac{\partial f}{\partial \theta}$. At each time t, we accumulate how the loss depends on parameters. The term $\frac{\partial f}{\partial \theta}$ is how parameters affect the dynamics, weighted by the adjoint $a(t)$.
Initial time gradient: $\frac{d}{dt}\frac{\partial L}{\partial t_0} = -a^T f$. This accounts for how changing

The Flow Map: Integral Form of Neural ODE

Definition: The flow map Φ integrates the ODE from t_0 to t_1:

$$z(t_1) = \Phi(z(t_0), t_0, t_1, \theta) = z(t_0) + \int_{t_0}^{t_1} f(z(\tau), \tau, \theta)\, d\tau \tag{14}$$

Key Properties:

- Φ is the exact solution operator of the ODE
- Depends on initial condition $z(t_0)$, time interval, and parameters θ
- Computed numerically via ODE solver (Euler, RK4, etc.)
- Differentiable with respect to all inputs!

The Loss Function:

$$L = L(z(t_1)) = L(\Phi(z(t_0), t_0, t_1, \theta)) \tag{15}$$

The Training Problem: Compute $\frac{dL}{d\theta}$ efficiently

Understanding the Flow Map: The flow map Φ is a central concept from dynamical systems theory. It maps initial conditions forward in time according to the differential equation.
Think of it like this: if you drop a leaf in a river at position $z(t_0)$ at time t_0, the flow map tells you where that leaf will be at time t_1. The river's current is the vector field $f(z, t, \theta)$.
Why "Flow"? Because the ODE defines a flow in state space. Particles flow along trajectories determined by $\frac{dz}{dt} = f(z, t, \theta)$.
The Integral Form: The equation $z(t_1) = z(t_0) + \int_{t_0}^{t_1} f(z(\tau), \tau, \theta)d\tau$ is the fundamental theorem of calculus applied to ODEs:

$$z(t_1) - z(t_0) = \int_{t_0}^{t_1} \frac{dz}{d\tau}d\tau = \int_{t_0}^{t_1} f(z(\tau), \tau, \theta)d\tau$$

Dependence on Parameters: The flow map depends on θ because the vector field f depends on θ. Changing θ changes the "currents" and thus where particles end up.
The Challenge: To train our neural network (optimize θ), we need $\frac{dL}{d\theta}$. But L depends on θ through this complex integral! We need to differentiate through the ODE solver.
Naive Approach (Bad): Backpropagate through every step of the ODE solver $\rightarrow$ O(N) memory where N = number of steps. For adaptive solvers, N could be 100s or 1000s!

The Gradient Challenge

What we want: $\frac{dL}{d\theta}$ where $L = L(\Phi(z(t_0), t_0, t_1, \theta))$

Chain rule attempt:

$$\frac{dL}{d\theta} = \frac{\partial L}{\partial z(t_1)} \frac{\partial z(t_1)}{\partial \theta} \tag{16}$$

The Problem: Computing $\frac{\partial z(t_1)}{\partial \theta}$ requires tracking how parameters affect the entire trajectory!

$$\frac{\partial z(t_1)}{\partial \theta} = \frac{\partial}{\partial \theta} \left[z(t_0) + \int_{t_0}^{t_1} f(z(\tau), \tau, \theta) d\tau \right] \tag{17}$$

This requires $\frac{\partial z(\tau)}{\partial \theta}$ for all $\tau \in [t_0, t_1]$ (the hidden states!)

The Core Issue

Can't pull $\frac{\partial}{\partial \theta}$ inside the integral because $z(\tau)$ itself depends on θ!

Why This is Hard: Let's be very explicit about the problem. We have:

$$L = L(z(t_1)) \quad \text{where} \quad z(t_1) = z(t_0) + \int_{t_0}^{t_1} f(z(\tau), \tau, \theta) d\tau$$

By the chain rule:

$$\frac{dL}{d\theta} = \frac{\partial L}{\partial z(t_1)} \frac{\partial z(t_1)}{\partial \theta}$$

The first term $\frac{\partial L}{\partial z(t_1)}$ is easy - we get this from autograd.
The second term $\frac{\partial z(t_1)}{\partial \theta}$ is the problem! Let's try to compute it:

$$\frac{\partial z(t_1)}{\partial \theta} = \frac{\partial}{\partial \theta} \int_{t_0}^{t_1} f(z(\tau), \tau, \theta) d\tau$$

By the chain rule inside the integral:

$$= \int_{t_0}^{t_1} \left[\frac{\partial f}{\partial z} \frac{\partial z(\tau)}{\partial \theta} + \frac{\partial f}{\partial \theta} \right] d\tau$$

But this contains $\frac{\partial z(\tau)}{\partial \theta}$ for all intermediate times τ! This is circular - to compute $\frac{\partial z(t_1)}{\partial \theta}$, we need $\frac{\partial z(\tau)}{\partial \theta}$ for all $\tau < t_1$.
Solution 1 (Bad): Store all $z(\tau)$ during forward pass, then backpropagate $\rightarrow$ O(N) memory.

Solution 2 (Good): Use Lagrange multipliers from optimal control theory!

Lagrange Multipliers: The Setup

Constrained Optimization: Minimize $L(z(t_1))$ subject to $\frac{dz}{dt} = f(z, t, \theta)$

Key Idea: Turn constraint into penalty using Lagrange multiplier $\lambda(t)$

$$\mathcal{L} = L(z(t_1)) - \int_{t_0}^{t_1} \lambda(t)^T \left[\frac{dz}{dt} - f(z(t), t, \theta)\right] dt \tag{18}$$

Why This Helps:

- When constraint is satisfied: $\frac{dz}{dt} = f$ so integral $= 0$
- Thus: $\mathcal{L} = L(z(t_1))$ (Lagrangian equals original loss)
- So: $\frac{d\mathcal{L}}{d\theta} = \frac{dL}{d\theta}$ (what we want!)
- But: We get to choose $\lambda(t)$ strategically!

The Strategy

Choose $\lambda(t)$ to eliminate the problematic $\frac{\partial z(\tau)}{\partial \theta}$ terms

Lagrange Multipliers Intuition: This is a classic technique from calculus of variations and optimal control theory. When we have a constraint, we can incorporate it into the objective function using a Lagrange multiplier.
The Key Insight: The constraint $\frac{dz}{dt} - f(z, t, \theta) = 0$ is always satisfied (it's our ODE!). So adding $-\int \lambda(t)^T[...]dt$ doesn't change the value of our objective - it's always zero!
But it DOES change the derivative! And we can choose $\lambda(t)$ to make the derivative computation easier.
Think of it Like This: We're rewriting our problem in a way that makes gradient computation tractable. It's like choosing a coordinate system - the physics doesn't change, but the math becomes easier.
Historical Context: This technique comes from Pontryagin's Maximum Principle in optimal control theory (1950s). The Lagrange multiplier $\lambda(t)$ will turn out to be the "adjoint state" or "costate" in control theory language.
What Makes This Work: The freedom to choose $\lambda(t)$ is powerful. We'll choose it so that when we compute $\frac{d\mathcal{L}}{d\theta}$, all the nasty terms involving $\frac{\partial z}{\partial \theta}$ cancel out!

This is the magic of the adjoint method.

Deriving the Adjoint Equation (Step 1)

Expand the Lagrangian:

$$\mathcal{L} = L(z(t_1)) - \int_{t_0}^{t_1} \lambda(t)^T \frac{dz}{dt} dt + \int_{t_0}^{t_1} \lambda(t)^T f(z(t), t, \theta) dt \tag{19}$$

Integration by Parts on the middle term:

$$\int_{t_0}^{t_1} \lambda(t)^T \frac{dz}{dt} dt = \lambda(t_1)^T z(t_1) - \lambda(t_0)^T z(t_0) - \int_{t_0}^{t_1} \frac{d\lambda}{dt}^T z(t) dt \tag{20}$$

Substitute back:

$$\mathcal{L} = L(z(t_1)) - \lambda(t_1)^T z(t_1) + \lambda(t_0)^T z(t_0) \tag{21}$$

$$+ \int_{t_0}^{t_1} \left[\frac{d\lambda}{dt}^T z(t) + \lambda(t)^T f(z(t), t, \theta) \right] dt \tag{22}$$

Integration by Parts Review: Recall that $\int u\,dv = uv - \int v\,du$.
For $\int_{t_0}^{t_1} \lambda(t)^T \frac{dz}{dt} dt$, set:

- $u = \lambda(t)^T$, so $du = \frac{d\lambda}{dt}^T dt$
- $dv = \frac{dz}{dt} dt$, so $v = z(t)$

Then:

$$\int_{t_0}^{t_1} \lambda(t)^T \frac{dz}{dt} dt = [\lambda(t)^T z(t)]_{t_0}^{t_1} - \int_{t_0}^{t_1} \frac{d\lambda}{dt}^T z(t) dt$$

Why Do This? Integration by parts moves the derivative from z to λ. This will let us choose how λ evolves!
The Boundary Terms: The terms $\lambda(t_1)^T z(t_1)$ and $\lambda(t_0)^T z(t_0)$ will be important. Notice we have $L(z(t_1))$ and $-\lambda(t_1)^T z(t_1)$ - we'll use this to set $\lambda(t_1)$!
The Integral Term: Now we have $\frac{d\lambda}{dt}$ in the integral. We get to choose how λ evolves in time by choosing $\frac{d\lambda}{dt}$!
This is setting up for the clever choice that makes everything work.

Deriving the Adjoint Equation (Step 2)

Take derivative with respect to θ:

$$\frac{d\mathcal{L}}{d\theta} = \frac{\partial L}{\partial z(t_1)}\frac{\partial z(t_1)}{\partial \theta} - \lambda(t_1)^T\frac{\partial z(t_1)}{\partial \theta} + \lambda(t_0)^T\frac{\partial z(t_0)}{\partial \theta} \tag{23}$$

$$+ \int_{t_0}^{t_1}\left[\frac{d\lambda}{dt}^T\frac{\partial z}{\partial \theta} + \lambda(t)^T\left(\frac{\partial f}{\partial z}\frac{\partial z}{\partial \theta} + \frac{\partial f}{\partial \theta}\right)\right]dt \tag{24}$$

Group terms with $\frac{\partial z}{\partial \theta}$:

$$\frac{d\mathcal{L}}{d\theta} = \left(\frac{\partial L}{\partial z(t_1)} - \lambda(t_1)^T\right)\frac{\partial z(t_1)}{\partial \theta} \tag{25}$$

$$+ \int_{t_0}^{t_1}\left[\left(\frac{d\lambda}{dt}^T + \lambda(t)^T\frac{\partial f}{\partial z}\right)\frac{\partial z}{\partial \theta}\right]dt \tag{26}$$

$$+ \int_{t_0}^{t_1}\lambda(t)^T\frac{\partial f}{\partial \theta}dt \tag{27}$$

Note: $z(t_0)$ is fixed, so $\frac{\partial z(t_0)}{\partial \theta} = 0$

Chain Rule Applied: When taking $\frac{d}{d\theta}$ of $f(z(t), t, \theta)$, we use:

$$\frac{d}{d\theta} f(z(t), t, \theta) = \frac{\partial f}{\partial z}\frac{\partial z}{\partial \theta} + \frac{\partial f}{\partial \theta}$$

The first term is problematic - it contains $\frac{\partial z}{\partial \theta}$ which depends on the entire trajectory!

Collecting Terms: Look at all the terms with $\frac{\partial z}{\partial \theta}$:

- Boundary: $\left(\frac{\partial L}{\partial z(t_1)} - \lambda(t_1)^T\right)\frac{\partial z(t_1)}{\partial \theta}$
- Integral: $\left(\frac{d\lambda}{dt}^T + \lambda(t)^T\frac{\partial f}{\partial z}\right)\frac{\partial z}{\partial \theta}$

The Clean Term: The last term $\int \lambda(t)^T \frac{\partial f}{\partial \theta} dt$ doesn't contain $\frac{\partial z}{\partial \theta}$! This is the gradient we can actually compute!

The Strategy: We want to make all the terms with $\frac{\partial z}{\partial \theta}$ vanish by choosing $\lambda(t)$ appropriately. Then only the clean term remains!

This is the key insight of the adjoint method.

The Adjoint Solution: Making Terms Vanish

Choose $\lambda(t)$ to eliminate all $\frac{\partial z}{\partial \theta}$ terms!

Boundary condition at t_1:

$$\lambda(t_1)^T = \frac{\partial L}{\partial z(t_1)} \quad \Rightarrow \quad \text{Boundary term vanishes!} \tag{28}$$

Dynamics of $\lambda(t)$ (adjoint equation):

$$\frac{d\lambda}{dt}^T = -\lambda(t)^T \frac{\partial f}{\partial z} \quad \Rightarrow \quad \text{Integral term vanishes!} \tag{29}$$

Final Gradient (what remains):

$$\frac{dL}{d\theta} = \int_{t_0}^{t_1} \lambda(t)^T \frac{\partial f}{\partial \theta} dt = -\int_{t_1}^{t_0} \lambda(t)^T \frac{\partial f}{\partial \theta} dt \tag{30}$$

Key Result

$\lambda(t) =$ adjoint state, solve backward from t_1 to t_0, then compute gradient!

The Boundary Choice: Setting $\lambda(t_1)^T = \frac{\partial L}{\partial z(t_1)}$ makes the boundary term:

$$\left(\frac{\partial L}{\partial z(t_1)} - \lambda(t_1)^T\right)\frac{\partial z(t_1)}{\partial \theta} = 0$$

vanish identically!

The Dynamics Choice: Setting $\frac{d\lambda}{dt}^T = -\lambda(t)^T\frac{\partial f}{\partial z}$ makes the integral term:

$$\left(\frac{d\lambda}{dt}^T + \lambda(t)^T\frac{\partial f}{\partial z}\right)\frac{\partial z}{\partial \theta} = 0$$

vanish identically!

What's Left: Only the "clean" term remains:

$$\frac{dL}{d\theta} = \int_{t_0}^{t_1} \lambda(t)^T\frac{\partial f}{\partial \theta}dt$$

The negative sign comes from reversing integration direction: $\int_{t_1}^{t_0} = -\int_{t_0}^{t_1}$.

Why This is Brilliant:

1. We never need to compute or store $\frac{\partial z}{\partial \theta}$ for any time!
2. We only need $\lambda(t)$, which we get by solving ONE ODE backward
3. We only need $\frac{\partial f}{\partial \theta}$, which we get from autograd for FREE!

Memory: O(1) because we only store current $\lambda(t)$ and $z(t)$, not the entire

How Neural ODEs Automate This

The Traditional Way (Impossible):

1. Derive adjoint equations by hand for your specific f
2. Compute Jacobians $\frac{\partial f}{\partial z}$ and $\frac{\partial f}{\partial \theta}$ analytically
3. Implement custom backward pass

The Neural ODE Way (Automatic):

1. Define $f(z, t, \theta)$ as a neural network in PyTorch/JAX
2. **Autodiff gives you $\frac{\partial f}{\partial z}$ and $\frac{\partial f}{\partial \theta}$ for FREE!**
3. Solve adjoint ODE using same ODE solver, but backward

The Magic of Autodiff

Vector-Jacobian Products (VJPs):
$\lambda(t)^T \frac{\partial f}{\partial z}$ and $\lambda(t)^T \frac{\partial f}{\partial \theta}$
computed via reverse-mode autodiff without forming full Jacobian!

Complexity: VJP costs $\approx$ 2-3$\times$ forward pass (not O(d^2) for Jacobian!)

Why Hand-Derivation is Impossible: For a neural network with millions of parameters and multiple layers, computing $\frac{\partial f}{\partial z}$ and $\frac{\partial f}{\partial \theta}$ by hand is impossible.
Even if we could, we'd need to implement custom code for every network architecture. This doesn't scale.
How Autodiff Saves Us: Modern autodiff systems (PyTorch, JAX, TensorFlow) can compute:

- **Forward mode:** Jacobian-vector products (JVPs): $\frac{\partial f}{\partial z} v$
- **Reverse mode:** Vector-Jacobian products (VJPs): $v^T \frac{\partial f}{\partial z}$

For the adjoint method, we need VJPs: $\lambda(t)^T \frac{\partial f}{\partial z}$ and $\lambda(t)^T \frac{\partial f}{\partial \theta}$.
Computational Cost: Computing $v^T \frac{\partial f}{\partial z}$ costs about as much as computing f itself (plus some overhead). This is MUCH cheaper than computing the full Jacobian matrix $\frac{\partial f}{\partial z}$ which would cost $\mathcal{O}(d^2)$ for d-dimensional z.
The `torchdiffeq` Implementation:

1. Forward: Solve $\frac{dz}{dt} = f(z, t, \theta)$ from t_0 to t_1
2. Compute loss $L(z(t_1))$
3. Initialize $\lambda(t_1) = \frac{\partial L}{\partial z(t_1)}$ (from autograd)
4. Backward: Solve augmented system backward using VJPs from autograd
5. Accumulate gradient: $\frac{dL}{d\theta} = -\int \lambda(t)^T \frac{\partial f}{\partial \theta} dt$

Adjoint Method: Why $\mathcal{O}(1)$ Memory?

Standard Backprop through ODE Solver:

- Store all intermediate states $h(t_i)$ for $i = 1, \dots, N$
- N depends on adaptive step size (could be 100s or 1000s)
- Memory: $\mathcal{O}(N)$ where $N =$ number of function evaluations

Adjoint Method:

- Only store final state $h(t_1)$
- During backward pass, recompute $h(t)$ as needed
- Memory: $\mathcal{O}(1)$ – just the current state!

Trade-off

Memory $\mathcal{O}(1)$ but computation $\approx 2\times$ (one forward, one backward solve)

The Memory Problem Explained: When you call PyTorch's autograd through an ODE solver, it builds a computation graph with a node for every function evaluation. For adaptive solvers with tight tolerances, this could be hundreds or thousands of nodes. Each node stores activations for backprop. For deep networks or long time horizons, this becomes prohibitive.

How Adjoint Saves Memory: Instead of storing the entire forward trajectory, we: 1. Run forward pass and save only $h(t_1)$ (the endpoint) 2. During backward pass, run the ODE solver again going backward from t_1 to t_0 3. Whenever we need $h(t)$ to compute Jacobians, we recompute it on-the-fly

This is called "checkpointing" or "recomputation": trade computation for memory.

Why Only $\mathcal{O}(1)$? At any moment during the backward pass, we only need: (1) current adjoint state $a(t)$, (2) current hidden state $h(t)$, (3) accumulated gradient $\frac{\partial L}{\partial \theta}$. All are $\mathcal{O}(d)$ where d is hidden dimension. The number of ODE solver steps N doesn't affect memory!

Computational Cost: We solve the ODE twice: once forward, once backward. Each costs roughly the same. So total compute is $\approx 2\times$ a forward pass. But memory is constant! For very deep networks (large N), this trade-off is worth it.

Practical Impact: You can train Neural ODEs with arbitrarily tight solver

Hyperparameter Selection

ODE Solver Tolerance

- `rtol`, `atol`: Control accuracy
- Higher tolerance $\rightarrow$ faster but less accurate
- Typical: `rtol=1e-3`, `atol=1e-4`

Solver Method

- **Adaptive**: 'dopri5', 'adams' (recommended)
- **Fixed-step**: 'euler', 'rk4' (for debugging)

Integration Time

- Usually $T = 1.0$ (can be learned)
- Longer $T \rightarrow$ more expressive but slower

Tolerance parameters: These control the error tolerance of the adaptive ODE solver. Think of them as a knob between "very accurate but slow" and "fast but approximate."

rtol (relative tolerance): Error relative to the magnitude of the solution. If $\texttt{rtol} = 10^{-3}$ and $|h(t)| \approx 1$, we allow errors $\sim 10^{-3}$.

atol (absolute tolerance): Minimum error threshold regardless of magnitude. Important when $h(t)$ is small.

The solver adapts step sizes to keep local error below: $\texttt{atol} + \texttt{rtol} \times |h(t)|$.

Choosing tolerances: Start with `rtol=1e-3`, `atol=1e-4`. If training is unstable, decrease them (slower but more accurate). If too slow, increase them. The key insight: tolerances affect both forward pass accuracy and gradient quality!

Solver methods: Adaptive methods (dopri5, adams) automatically adjust step size. Fixed-step methods (euler, rk4) take uniform steps – useful for debugging but inefficient for training.

Integration time: We integrate from $t = 0$ to $t = T$. Larger T gives the dynamics more "time" to transform the input, increasing capacity. But: longer integration $\rightarrow$ more function evaluations $\rightarrow$ slower training. Neural ODEs typically use $T = 1$, but this can be a learnable parameter!

Latent ODEs for Irregular Time Series

Challenge: Irregular, sparse observations with variable time gaps

Approach: Combine RNNs and ODEs

- **Encoder (ODE-RNN):** Process observations backward in time
- **Latent ODE:** Smooth dynamics in continuous time
- **Decoder:** Generate predictions at any time

Key Innovation: Poisson process prior for observation times

$$p(t_1, \ldots, t_N | z_0) = \prod_{i=1}^{N} \lambda(t_i | z_0) \exp\left(-\int_0^T \lambda(t|z_0) dt\right) \tag{31}$$

Models *when* observations occur, not just their values.

The Problem: Real-world time series (medical records, sensor data) are irregularly sampled. Standard RNNs assume fixed time steps. Naive solutions like interpolation or resampling lose information.

ODE-RNN Encoder: Process observations in reverse chronological order. At each observation, (1) compute ODE dynamics backward to previous observation, (2) update hidden state with RNN cell. This produces a latent representation that accounts for variable time gaps.

Latent ODE Dynamics: In the latent space, use a Neural ODE to model smooth, continuous dynamics. This allows querying the state at arbitrary times, not just at observation points.

VAE Framework: Treat this as a variational autoencoder. The encoder (ODE-RNN) produces $q(z_0|\text{observations})$, the latent ODE is the generative model $p(z(t)|z_0)$, and we optimize the ELBO:

$$\mathcal{L} = \mathbb{E}_{q(z_0)}[\log p(\text{data}|z)] - \text{KL}(q(z_0)\|p(z_0))$$

Poisson Process for Observation Times: A brilliant addition! Instead of treating observation times as fixed, model them with intensity $\lambda(t|z_0)$. This captures that observations might be more likely when the system is in certain states (e.g., hospital visits when patient condition changes). The integral $\int \lambda(t)dt$ is the expected number of observations – we can compute this via ODE!

This approach handles: missing data, irregular sampling, variable-length

Latent ODE Architecture Details

Three-Component System

1. ODE-RNN Encoder (Backward)

Process observations $\{(t_i, x_i)\}_{i=1}^{N}$ in *reverse* time order:

$$h_i = \mathrm{ODESolve}(h_{i+1}, t_{i+1} \to t_i) \quad \text{then} \quad h_i \leftarrow \mathrm{RNN}(h_i, x_i)$$

Output: Initial latent state $z_0 \sim q(z_0 | x_{1:N})$

2. Latent ODE Dynamics

Continuous evolution in latent space:

$$\frac{dz(t)}{dt} = f_\theta(z(t), t)$$

3. Decoder

Map latent states to observations: $p(x_i | z(t_i))$

VAE Framework: This is trained as a variational autoencoder:
Encoder: $q_\phi(z_0|x_{1:N})$ produces a distribution over initial states (typically Gaussian with learned mean and variance)
Prior: $p(z_0) = \mathcal{N}(0, I)$ (standard normal)
Generative Model:

1. Sample $z_0 \sim p(z_0)$
2. Evolve via ODE to get $z(t_i)$ for each observation time
3. Generate observations $x_i \sim p(x|z(t_i))$

Training Objective (ELBO):

$$\mathcal{L} = \mathbb{E}_{q(z_0)}[\sum_{i=1}^{N} \log p(x_i|z(t_i))] - \mathrm{KL}(q(z_0)\|p(z_0))$$

The first term is reconstruction: how well can we predict observations from latent states? The second term is regularization: keep the latent distribution close to the prior.
Key Advantage: The KL term is computed only on z_0, not on the entire trajectory. The ODE deterministically maps z_0 to all future $z(t)$, so we only need uncertainty about the initial state!

1. **Why Process Backward?** This is a key design choice! Processing observations backward in time naturally produces an initial condition z_0 that encodes the entire sequence. Think of it like

Training Latent ODEs: The ELBO

Evidence Lower Bound (ELBO): Variational inference objective

$$\mathcal{L}_{\text{ELBO}} = \underbrace{\mathbb{E}_{q(z_0|x)}[\sum_{i=1}^{N} \log p(x_i|z(t_i))]}_{\text{Reconstruction}} - \underbrace{D_{KL}(q(z_0|x)\|p(z_0))}_{\text{Regularization}} \tag{32}$$

Component 1: Reconstruction Loss

- Measures how well the model predicts observations
- $q(z_0|x)$: Encoder's posterior over initial state
- $z(t_i)$: Latent state at time t_i via ODE
- Typically Gaussian: $\log p(x_i|z(t_i)) = \log \mathcal{N}(x_i|\mu(z(t_i)), \sigma^2)$

Component 2: KL Divergence

- Regularizes latent space to match prior $p(z_0) = \mathcal{N}(0, I)$
- Prevents overfitting and ensures smooth latent space
- Only computed at $t = 0$, not entire trajectory!

What is ELBO? The Evidence Lower Bound is the objective function for variational autoencoders (VAEs). We can't directly optimize the data likelihood $p(x)$ because it requires integrating over all possible latent states. Instead, we introduce an approximate posterior $q(z_0|x)$ and maximize a lower bound.

Derivation Sketch:

$$\begin{aligned}\log p(x) &= \log \int p(x|z_0)p(z_0)dz_0 \\ &\geq \mathbb{E}_{q(z_0|x)}[\log p(x|z_0)] - D_{KL}(q(z_0|x)\|p(z_0))\end{aligned}$$

The inequality comes from Jensen's inequality. Maximizing the ELBO is equivalent to minimizing the gap between $q(z_0|x)$ and the true posterior $p(z_0|x)$. **Why Two Terms?**

Reconstruction Term: $\mathbb{E}_{q(z_0|x)}[\sum_{i=1}^{N} \log p(x_i|z(t_i))]$

This says: "Sample z_0 from the encoder, evolve it via the ODE to get $z(t_i)$ at each observation time, then decode to get predictions. How likely are the actual observations under these predictions?"

In practice:

1. Encoder outputs μ_{z_0}, σ_{z_0}
2. Sample $z_0 \sim \mathcal{N}(\mu_{z_0}, \sigma_{z_0}^2)$ (reparameterization trick)
3. Solve ODE: $z(t_i) = z_0 + \int_0^{t_i} f_\theta(z(\tau), \tau)d\tau$

Latent ODE: Handling Observation Times

Innovation: Model *when* observations occur, not just their values

Poisson Process Intensity:

$$\lambda(t|z_0) = g_\psi(z(t)) \tag{33}$$

where g_ψ is a neural network mapping latent states to observation rates.

Joint Likelihood:

$$p(\{x_i, t_i\}_{i=1}^N | z_0) = \underbrace{\prod_{i=1}^N p(x_i|z(t_i))}_{\text{observations}} \times \underbrace{\prod_{i=1}^N \lambda(t_i|z_0) \exp\left(-\int_0^T \lambda(t|z_0)dt\right)}_{\text{timing}} \tag{34}$$

The integral $\int_0^T \lambda(t|z_0)dt$ is computed by solving an ODE!

Why Model Observation Times? In many applications, *when* we observe data is informative:

- Medical data: patients visit hospital more often when sick
- Sensor data: measurements may be triggered by events
- Social media: posting frequency varies with user state

Treating observation times as fixed throws away this information!
Poisson Process Basics: A Poisson process is characterized by an intensity function $\lambda(t)$. The probability of an event in a small interval $[t, t+dt]$ is $\lambda(t)dt$.
The likelihood of observing events at times $t_1, \ldots, t_N$ over $[0, T]$ is:

$$p(t_1, \ldots, t_N) = \prod_i \lambda(t_i) \times \exp\left(-\int_0^T \lambda(t)dt\right)$$

The first term: probability of events at the observed times. The second term: probability of no events at all other times.
Making λ State-Dependent: We let $\lambda(t) = g(z(t))$ depend on the latent state. This captures that observation rate depends on system state. For example, $z(t)$ represents patient health, and $\lambda(t)$ is hospital visit rate.
Computing the Integral: We need $\int_0^T g(z(t))dt$. Define a new variable $\Lambda(t) = \int_0^t g(z(s))ds$. Then:

Function Encoders with Neural ODEs

Goal: Transfer learned dynamics to new systems without gradient updates

Approach: Learn a basis of dynamics

1. Learn K basis ODEs: $\frac{dz_i}{dt} = f_i(z_i, t)$ for $i = 1, \ldots, K$
2. For new system: encode demonstrations $\rightarrow$ coefficients α_i
3. Predict: $\frac{dz}{dt} = \sum_{i=1}^{K} \alpha_i f_i(z, t)$

Key Idea: Treat dynamics as vectors in a Hilbert space
Any trajectory $x(z, t)$ can be represented as: $x \approx \sum_{i=1}^{K} \alpha_i \phi_i$

Zero-Shot Transfer

Compute coefficients α_i from demonstrations without any gradient updates!

Hilbert Space Setup: We need to define an inner product between dynamical systems. For functions, $\langle f, g \rangle = \int f(x)g(x)dx$. For dynamics? A dynamical system is characterized by its velocity field $v(z,t) = \frac{dz}{dt}$. We define the inner product as:

$$\langle v_1, v_2 \rangle = \int_0^T \int_{\mathcal{Z}} v_1(z,t) \cdot v_2(z,t)\, p(z,t)\, dz\, dt$$

where $p(z,t)$ is a measure over states and time.
For Neural ODE Basis: The basis function ϕ_i has velocity field $f_i(z,t)$. To compute the coefficient for a demonstration trajectory x_{demo}:

$$\alpha_i = \langle \phi_i, x_{\text{demo}} \rangle = \mathbb{E}_{t,z \sim \phi_i}[v_{\text{demo}}(z,t) \cdot f_i(z,t)]$$

where $v_{\text{demo}}(z,t)$ is the velocity of the demonstration at state z and time t.
Monte Carlo Computation:

1. Run the basis ODE ϕ_i to generate trajectory (t_j, z_j) samples
2. At each sample point, evaluate the demonstration's velocity: $v_{\text{demo}}(z_j, t_j)$
3. Compute dot product with basis velocity: $v_{\text{demo}}(z_j, t_j) \cdot f_i(z_j, t_j)$
4. Average over all samples: $\alpha_i \approx \frac{1}{M} \sum_{j=1}^{M} v_{\text{demo}}(z_j, t_j) \cdot f_i(z_j, t_j)$

This is efficient because we only need forward passes through f_i, no back-

Function Encoder: Implementation Details

Computing Demonstration Velocity: Given demonstration points $\{(t_j, z_j)\}$:

- **Direct differentiation:** If demonstration is a continuous function, compute $\frac{dz}{dt}$ numerically
- **Finite differences:** For discrete observations: $v(t_j) \approx \frac{z_{j+1}-z_j}{t_{j+1}-t_j}$

Inner Product Formula:

$$\alpha_i = \mathbb{E}_{t\sim\mathcal{U}[0,T], z\sim z_i(t)}\Big[\underbrace{v_{\text{demo}}(z,t)}_{\text{demo velocity}} \cdot \underbrace{f_i(z,t)}_{\text{basis velocity}}\Big] \quad (35)$$

Key Properties:

- $\alpha_i > 0$: demonstration aligns with basis i
- $\alpha_i < 0$: demonstration opposes basis i
- $|\alpha_i|$ large: basis i is important for this system

Why This Works: The inner product measures similarity between two velocity fields. If the demonstration tends to move in the same direction as basis function i, then α_i will be large.

Handling Discrete Demonstrations: In practice, we don't have continuous functions. We have discrete observations $(t_1, z_1), (t_2, z_2), \ldots$. To compute velocities:

Method 1 (Finite Differences):

$$v_{\text{demo}}(t_j) = \frac{z_{j+1} - z_j}{t_{j+1} - t_j}$$

Simple but can be noisy, especially if observations are irregularly spaced.

Method 2 (Neural Network Fit): Train a small network to fit $z(t)$ from demonstrations, then differentiate the network. More robust to noise.

Method 3 (Latent ODE Approach): Use the latent ODE framework to infer a smooth latent trajectory, then compute velocities from the latent dynamics.

Computing the Expectation: The expectation is over (t, z) pairs sampled from the basis ODE trajectory. In practice:

1. Solve basis ODE ϕ_i to get trajectory: $(t_1, z_1), \ldots, (t_M, z_M)$
2. At each point, evaluate demo velocity (via interpolation or nearest neighbor)
3. Compute $f_i(z_j, t_j)$ (just a forward pass through the basis network)

Extensions

Augmented Neural ODEs

Add extra dimensions to avoid topological constraints

Second-order Neural ODEs

Include acceleration: $\frac{d^2h}{dt^2} = f(h, \frac{dh}{dt}, t)$

Stochastic Differential Equations (SDEs)

Add noise for uncertainty: $dh = f(h, t)dt + g(h, t)dW$

Hamiltonian Neural Networks

Preserve energy and symplectic structure

Augmented Neural ODEs: Standard Neural ODEs have a limitation: they define continuous flows that cannot cross themselves. This means they cannot solve certain problems (like learning a spiral that crosses itself). Solution: augment the state space with extra dimensions. Instead of $h \in \mathbb{R}^d$, use $[h, z] \in \mathbb{R}^{d+p}$ where z are auxiliary variables initialized to zero. These extra dimensions give the flow more "room to maneuver."

Second-order Neural ODEs: Some physical systems naturally involve second derivatives (e.g., Newton's laws: $F = ma = m\frac{d^2x}{dt^2}$). Instead of modeling velocity, we model acceleration. Define state as $s = [h, v]$ where $v = \frac{dh}{dt}$. Then: $\frac{dh}{dt} = v$ and $\frac{dv}{dt} = f(h, v, t, \theta)$. This is useful for mechanical systems and can improve training dynamics.

Neural SDEs: Add stochastic noise to model uncertainty or learn stochastic processes. The equation $dh = f(h, t)dt + g(h, t)dW$ includes a drift term f (deterministic) and diffusion term g (stochastic), where dW is Brownian motion. Useful for generative modeling and Bayesian inference. The latent ODE paper uses this for modeling irregular time series.

Hamiltonian Neural Network: This is one of the most powerful ideas: you can swap out your integrator to add physical structure!

For physical systems, we want to preserve conservation laws (energy, momentum). Instead of learning f directly, learn the Hamiltonian $H(q, p)$ (total energy), then compute:

$$\frac{dq}{} = \frac{\partial H}{} \qquad \frac{dp}{} = -\frac{\partial H}{}$$

Practical Tip: Neural ODEs + Interpretable Models

Problem: Neural ODEs are powerful but not interpretable
$f(x, t, \theta)$ is a black-box neural network – can't extract equations!

Solution: Use Neural ODEs as a preprocessing step

1. Train Neural ODE on irregular/noisy data
2. Generate clean, regularly-spaced data from trained Neural ODE
3. Pass regular data to interpretable method (SINDy, symbolic regression)

Best of Both Worlds

Neural ODE: handles irregular data, noise robust, accurate
SINDy/Symbolic: gives interpretable equations like $\dot{x} = \mu x - x^3$

Neural ODEs and methods like SINDy/symbolic regression are complementary!
When SINDy Fails: SINDy (Sparse Identification of Nonlinear Dynamics) and other equation discovery methods work best with:

- Clean data
- Regularly spaced measurements
- Low noise

But real data is often: irregular, noisy, missing observations.
The Pipeline:

1. **Step 1:** Train Neural ODE on your messy, irregular data
2. **Step 2:** Use trained Neural ODE to generate synthetic data
   - Choose regular time spacing
   - Dense sampling
   - Clean trajectories
3. **Step 3:** Apply SINDy/symbolic regression to synthetic data
4. **Step 4:** Get interpretable equations!

Example: Irregular sensor data $\rightarrow$ Neural ODE $\rightarrow$ regular synthetic data $\rightarrow$ SINDy discovers: $\dot{x} = -0.1x + 2y^3$, $\dot{y} = -2x^3 - 0.1y$
Why This Works: Neural ODE acts as a "denoiser" and "regularizer"

Summary

Key Takeaways

1. Neural ODEs = continuous-depth neural networks
2. ResNets are crude Euler integration; Neural ODEs use better solvers
3. Adjoint method enables $\mathcal{O}(1)$ memory training via autodiff
4. Works with irregular time series (unlike ResNets/RNNs)
5. Can bake in physical structure (Hamiltonian, Lagrangian, symplectic)
6. Applications: classification, time series, generative models, physics

The Big Idea

Learn the **vector field** (continuous dynamics), not discrete transformations

Key advantage: Leverage 300+ years of ODE theory and numerical methods!

References

Chen, R. T. Q., Rubanova, Y., Bettencourt, J., & Duvenaud, D. (2018).
Neural Ordinary Differential Equations.
NeurIPS 2018 (Best Paper Award).

Grathwohl, W., Chen, R. T. Q., Bettencourt, J., Sutskever, I., & Duvenaud, D. (2019).
FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models.
ICLR 2019.

Rubanova, Y., Chen, R. T. Q., & Duvenaud, D. (2019).
Latent ODEs for Irregularly-Sampled Time Series.
NeurIPS 2019.